Final Technical Report
An Extensible and Scalable Framework for Formal Modeling, Analysis, and Development of Distributed Systems
Contract No. FA9550-07-C-0114
STTR Phase I

Nancy A. Lynch* Laurent D. Michel† Alexander A. Shvartsman‡

November 30, 2008

Project summary. This Small Business Technology Transfer Phase I project advanced the state of the art in formal modeling and engineering of complex distributed systems. The pursued research and development approach included: (a) modeling language(s) that can be used to represent complex distributed systems and their components, theory and methodology providing sound mathematical basis for modeling systems and reasoning about their properties, (b) extensible and scalable analysis tools that can be used to validate correctness and performance properties, and automated synthesis tools that can produce efficient deployment schemes of the software components in target networks subject to specified constraints.

In prior research we developed an expressive modeling language, called Tempo, that can be used to represent complex distributed systems and their components. We also have developed the theory and methodology providing sound mathematical basis for modeling systems and reasoning about their properties, along with tools that can be used to validate correctness and performance properties. In this project we have extended the methodology to incorporate additional means for reasoning about probabilistic and hybrid systems. Based on these developments, we built an extended integrated development environment, called Tempo, for modeling, synthesis, and analysis of distributed systems We have also prototyped a methodology that can be used to generate working code from system models and yield efficient deployment of the software components in target networks. The ultimate goal of this work is to develop a complete and comprehensive formal methodology based in sound theory, and an industrial-grade extensible integrated software engineering environment for the implementors of modern distributed software systems. The preliminary release of the system for Linux, Windows, and OSX-PPC platforms is available at www.veromodo.com.

*Co-Principal Investigator and Chief Technology Officer, VeroModo, Inc.. Email: lynch@theory.csail.mit.edu.
†Principal Investigator. Email: ldm@cse.uconn.edu.
‡Co-Principal Investigator and President, VeroModo, Inc.. Email: aas@veromodo.com.

Contents

1 Introduction **3**
- 1.1 The problem and our solution . . . 3
- 1.2 Summary of project objectives and accomplishments . . . 4
- 1.3 Project team and academic partner institution . . . 6
- 1.4 Publications . . . 6
- 1.5 Acknowledgments . . . 7
- 1.6 Document structure . . . 7

2 Tempo Toolkit for Timed Input/Output Automata Formalism **7**
- 2.1 What is the Tempo language? . . . 7
- 2.2 Tempo language overview . . . 8
  - 2.2.1 Timed I/O Automata . . . 8
  - 2.2.2 The Tempo language and tools . . . 9
- 2.3 An Example: mutual exclusion algorithm . . . 9
  - 2.3.1 The Tempo specification . . . 10
  - 2.3.2 Properties of the algorithm . . . 14

3 Extensions of I/O Automata and Timed I/O Automata Frameworks **14**
- 3.1 Probabilistic extensions . . . 15
- 3.2 Extensions for reasoning about security protocols . . . 15
- 3.3 Extensions for hybrid systems . . . 15

4 Tempo Toolkit: Architecture and Language **16**
- 4.1 The Architecture of Tempo . . . 17
- 4.2 Tempo language . . . 18

5 Deployment Problems **18**
- 5.1 Augmenting Tempo with deployment annotations . . . 19
- 5.2 Language Extensions for Deployment Annotations . . . 21
- 5.3 Eventually Serializable Data Service Annotations . . . 25

6 Generating deployment models **26**
- 6.1 Translation Scheme . . . 26
- 6.2 Comet Program . . . 28
- 6.3 Tempo Language Restrictions . . . 29

7 Solving deployment models **29**
- 7.1 The Abstract Model . . . 30
- 7.2 The CP Model . . . 30
  - 7.2.1 The Model . . . 31
  - 7.2.2 The Search Procedure . . . 32

8 A Formal Treatment of an Abstract Channel Implementation Using Java Sockets and TCP **33**
- 8.1 Rationale: towards code generation . . . 34
- 8.2 Technical development: channel implementation . . . 35

9 Conclusion **37**

Bibliography and References **38**

1 Introduction

This is the final technical report for Phase I STTR project that focused on extensions to the TIOA (Timed Input/Output Automata) language and framework as well as its companion system: Tempo. Specifically, it describes the advancements in the theory of TIOA and the addition of a language extension and software tool back-end aimed at assisting users of the methodology when turning their attention to the implementation aspects of their distributed system. The new extension makes it possible to specify the characteristics of a deployment environment at the TIOA level with model annotations. The annotations are then used by Tempo to derive a combinatorial optimization model that produces an optimal (with respect to an objective specified in the annotations) deployment scenario. The back-end tool relies on state-of-the-art combinatorial optimization tool to solve the optimization problem. The Tempo tool-chain now offers an end-to-end solution starting with the specification of a distributed algorithm to its optimal deployment on a target platform.

1.1 The problem and our solution

Challenges in developing distributed systems. Developing dependable distributed systems for modern computing platforms continues to be challenging. While the availability of distributed middleware makes feasible the construction of systems that run on distributed platforms, ensuring that the resulting systems satisfy specific safety, timing, and fault-tolerance requirements remains problematic. The middleware services used for constructing distributed software are specified informally and without precise guarantees of efficiency, timing, scalability, compositionality, and fault-tolerance. Even when services and algorithms are specified formally, rigorous reasoning about the specifications is often left out of the development process.

As contemporary distributed systems continue to grow in complexity and sophistication in many domains, these systems are required to have formally-specified guarantees of safety, performance, and fault-tolerance. Current software-engineering practice limits the specification of such requirements to informal descriptions. When formal specifications are given, they are typically provided only for the system interfaces. The specification of interfaces alone stops far short of satisfying the needs of users of critical systems. Such systems need to be equipped with precise specifications of their semantics and guaranteed behavior. When a system is built of smaller components, it is important to specify the properties of the system in terms of the properties of its components.

We view formal specification and analysis as valuable tools that should be at the disposal of the developers of distributed systems.

Furthermore, once a system is specified, the implementation efforts are often informally connected to the specification which implies that the guarantees offered by the formal tools may not carry over to the implementation. Additionally, the deployment of the implementation on an actual platform raises its own set of challenges to meet the timing, fault-tolerance and scalability requirements.

It is thus desirable to offer an integrated approach that covers the entire process, from design to implementation and deployment of the resulting distributed system.

Our approach to modeling and analysis of complex distributed systems. This project developed techniques and tools that are designed to be used in constructing provably correct distributed software. At the specification level, it leverages the IOA formalism (named after Input/Output Automata) and its companion toolset Tempo. IOA use mathematical models—in

particular, interacting state machines—as an integral part of the software development process. The stages of this process within the scope of our framework are as follows. Abstract requirements for a distributed system are specified using a modeling language. These specifications are then refined through multiple levels of abstraction. Each refinement step is formally validated. Validation techniques include a combination of simulation, model checking, and theorem proving. The goal of the refinement process is to produce sufficiently detailed models that (a) can ultimately be used to generate distributed code automatically, and that (b) are guaranteed to be consistent with the modeled system requirements.

To support automated formal methods for constructing or analyzing systems, a modeling language must rest on a solid mathematical foundation. The I/O automaton model [20] and its timed extensions [13] provide such a foundation. I/O automata have been used to describe and verify many distributed algorithms and systems (see, for example, [16]), and Timed I/O Automata have been used to model timing-dependent distributed algorithms and real-time control systems [13].

The Tempo language uses the Timed I/O Automata to describe interacting state machines. They are nondeterministic, which makes them suitable for describing systems in their most general forms. The state of a TIOA can change in two ways: *discrete transitions*, which are labeled by discrete actions, change the state instantaneously, whereas *trajectories* are functions that describe the evolution of the state variables over intervals of time.

Target systems. Many types of systems are currently developed using software engineering methodology that is less than adequate in its ability to handle formal modeling and analysis of complex distributed software, and we anticipate that several specific types of systems will benefit from being designed within our proposed framework. The types of systems include:

- Distributed data systems: data collection, management, dissemination; consistent replicated shared-data systems.
- Communication: group communication systems, broadcast and multicast systems with quality-of-service guarantees.
- Coordination and control: traffic management, industrial process control, automated manufacturing systems, transportation (e.g., TCAS, traffic collision avoidance system used in civil aviation).

Many such systems involve specialized distributed platforms, such as networks of sensors and mobile *ad hoc* networks.

1.2 Summary of project objectives and accomplishments

With the ultimate goal of providing a more complete formal methodology and associated tools to substantially improve the state of the art in developing software for complex distributed systems, the project objectives encompassed the following.

Theory and Methodology. In developing service definitions and algorithms for distributed systems, analyzing the resulting specification, and generating code from specifications for such systems, the results only make sense if they are ultimately based on a sound underlying mathematical model.

This project uses interacting state machine models. Standard models like I/O (Input/Output) automata and timed I/O automata provide the foundation for the Tempo language that we developed previously. However, some of the systems require richer models capable of describing complexities such as probabilistic and continuous behaviors. New models for particular kinds of timing and failure behavior and corresponding analysis methods are also needed. Existing models that handle such features need to be improved and better integrated. Other extensions are needed to express constraints for deploying systems defined in terms of the Tempo language in physical target networks, and for optimizing deployment over target networks based on various performance considerations. We collectively refer to these I/O automata models and the languages for system specifications in these models as *Tempo**. In this project we have advanced the theory for such models, in particular the probabilistic extensions, we have extended the formal framework and methodology for analyzing system specifications, and deriving distributed code from such specifications, and we have prototyped languages for specifying complex distributed systems in such models.

Our accomplishments in this area are presented in this report as follows. The probabilistic and hybrid extensions to the Input/Output Automata framework are presented in Section 3. The deployment-oriented augmentations of the Tempo framework are presented in Section 5.

Modeling, Analysis, Code Generation, and System Deployment Tools. We performed research and feasibility studies needed to develop computer-aided design tools for analysis of complex distributed systems expressed in the *Tempo** formalism and to prototype such tools on the basis of the Tempo framework developed previously. New modeling and analysis tools and tool extensions that are the result of our work include the following. The *language processor* is a front end tool that will accept *Tempo** specifications, perform static and type analysis, and produce intermediate output for use by other tools. The *simulator* is a tool designed to simulate executions of *Tempo** specifications and to provide linked simulations of pairs of specifications, where one specification gives an abstract definition and the other is a more concrete specification that is supposed to implement the abstract definition.

Building on our prior work on code generation for distributed systems, we have explored formal approaches to code generation from *Tempo** specifications, and prove theorems about the correctness of the resulting code. We prototyped tools for mapping *Tempo** system specifications consisting of multiple automata to target networks subject to distributed deployment constraints and efficiency and resource considerations, e.g., communication bandwidth, storage requirements, and redundancy for fault-tolerance.

Our accomplishments in this area are presented in this report as follows. An overview of and our latest refinements to the existing Tempo integrated development environment are presented in Sections 2 and 4. The tools and translators for dealing with deployment problem of systems specified in Tempo are presented in Sections 5 and 6. In Section 8 we summarize our work on formal treatment of channel implementations as a part of our work towards code generation extensions.

Applications: Evaluations and Feasibility. In order to evaluate the effectiveness, scalability, and extensibility of our methodology and prototypes, we applied them to model and analyze representative systems. Compared to previous attempts to optimize the deployment of interesting systems we have obtained substantial improvements using our integrated approach with constrained-programming based solutions.

We present out accomplishments in Section 6 and 7.

1.3 Project team and academic partner institution

The project team over the duration of the effort included the following people:

Laurent Michel, Ph.D., System Architect and PI
Nancy Lynch, Ph.D., Chief Technical Officer and Co-PI
Alex Shvartsman, Ph.D., Project Manager and Co-PI
Carleton Colfrin, Senior Software Engineer
Elaine Sonderegger, Graduate Researcher, Development
Dilsun Kaynar, Tempo Consultant

Our academic partner on this project was the University of Connecticut

1.4 Publications

In this section we list publications directly related to the project that were authored or co-authored by the project personnel. All publications are available on request.

[P1] R. Canetti, L. Cheung, D. Kaynar, M. Liskov, N. Lynch, O. Pereira, and R. Segala. Analyzing Security Protocol Using Time-Bounded Task-PIOAs. *Journal of Discrete Event Dynamic Systems (DEDS)*, Springer, volume 18, number 1, March 2008.

[P2] Ran Canetti, Ling Cheung, Dilsun Kaynar, Nancy Lynch, and Olivier Pereira. Modeling Bounded Computation in Long-Lived Systems. *CONCUR 2008, Concurrency Theory, 19th International Conference*, pages 114–130, 2008.

[P3] Chryssis Georgiou, Peter M. Musial, Alexander A. Shvartsman, Elaine L. Sonderegger: An Abstract Channel Specification and an Algorithm Implementing It Using Java Sockets. *Proceedings of The Seventh IEEE International Symposium on Networking Computing and Applications, NCA 2008*, pages 211–219, 2008.

[P4] Daniel Liberzon, Sayan Mitra, and Nancy Lynch. Verifying Average Dwell Time of Hybrid Systems. To appear in *ACM Transactions in Embedded Computing Systems.*

[P5] N. Lynch, L. Michel, and A. Shvartsman, "Tempo: A Toolkit for the Timed Input/Output Automata Formalism", *First International Conference on Simulation Tools and Techniques for Communications, Networks and Systems (SIMUTools 2008).* Industrial Track: Simulation Works. CDROM, paper 3105, 8 pages, Marseilles, France, March 4-7, 2008.

[P6] L. Michel, A. Shvartsman, E. Sonderegger and P. Van Hentenryck, "Optimal Deployment of Eventually-Serializable Data Services.", *Proceedings of the Fifth International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR 2008*, Paris, France, May 20-23, 2008.

[P7] L. Michel, A. Shvartsman, E. Sonderegger and P. Van Hentenryck "Optimal Deployment of Eventually-Serializable Data Services", Submitted to *Annals of Operations Research*, October, 2008.

(The complete bibliography cited in this report is included after the main text.)

1.5 Acknowledgments

The work described in this report was funded by a contract from AFOSR. We are grateful to AFOSR for this support, and we thank the Program Managers Dr. R. Herklotz and Dr. D. Luginbuhl for their constructive guidance and encouragement.

We thank our academic partner for substantial contributions to the success of the project. We thank all participants listed in the section above for their contributions to the project.

Earlier work on a prototype framework was performed at MIT by Anna Chefter, Stephen Garland, Dilsun Kaynar, Panayiotis Mavrommatis, Antonio Ramirez, and Edward Solovey.

1.6 Document structure

Section 2 presents the overall framework at a high level, including tools descriptions, and specification examples. In the rest of the report we describe in detail the Phase I work and our accomplishments. In Section 3 we describe the advances in extending the formal Input/Output Automata and Timed Input/Output Automata frameworks to include reasoning about probabilistic and hybrid systems. In Section 4 we first briefly review the architecture of the integrated Tempo framework and focus in section 5.1 on the new annotations related to deployment issues. In Section 6 we present the translation module responsible for deriving combinatorial optimization models from Tempo specifications. In Section 7 we discuss the optimizer back-end and illustrate its capabilities on the deployment of a distributed system (Eventually Serializable Data Services). In Section 8 we summarize our work on formal treatment of channel implementations as a part of our work towards code generation extensions.

We conclude in Section 9. Bibliography completes this report.

2 Tempo Toolkit for Timed Input/Output Automata Formalism

Tempo is a formal language for modeling distributed, concurrent, and timed systems as collections of interacting state machines, called timed input/output automata. Tempo provides natural mathematical notations for describing systems, their intended properties, and intended relationships between their descriptions at varying levels of abstraction. The Tempo Toolkit is an implementation of the Tempo language and a suite of tools that supports a range of validation methods for descriptions of systems and their properties, including static analysis, simulation, and machine-checked proofs. This section gives an overview of the Tempo language and illustrates its utility on selected examples of importance to distributed computing. The focus of the presentation is on the Tempo tools. We quickly review the purpose of Timed I/O Automata and TEMPO language 2.1, the TEMPO toolset (Section 2.2), and briefly review an example in Section 2.3.

2.1 What is the Tempo language?

Tempo is a formal language for modeling distributed systems as collections of interacting state machines called *Timed Input/Output Automata* [13]. Timed Input/Output Automata are often referred to as *Timed I/O Automata*, or just *TIOAs*. The distributed systems in question may have timing constraints, for example, bounds on the time when certain events may occur, or bounds on the rates of change of component clocks. They may use time in significant ways, for example, for timeouts, or for scheduling events to occur periodically. Timed I/O Automata formalism provides

good support for describing these constraints and capabilities. Timed and untimed I/O Automata formalisms have been effectively used for specifying numerous distributed and concurrent algorithms [16]. The Tempo language provides simple formal notation for describing Timed I/O Automata precisely, based on the pseudocode notation that has been used in many research papers. It also allows specification of properties such as invariant assertions and relationships between automata at different levels of abstraction. The Tempo language is supported by an associated integrated development environment toolkit, also called Tempo, that provides an extensible framework supporting a range of integrated analysis and validation tools, including static analysis, simulation, model-checking, and theorem-proving.

Many distributed systems involve a combination of computer components and real-world, physical entities such as vehicles, robots, or medical devices. Systems involving interaction between computer and real-world components usually have strong safety, reliability, and predictability requirements, stemming from the requirements of real-world applications. This makes it especially important to have good methods for modeling the systems precisely and analyzing their behavior rigorously. Tempo provides a simple, elegant, and powerful mathematical foundation for analyzing a wide variety of systems, and it can be used to model both computer and real-world system components, as well as their interactions.

Tempo can be used to model practically any type of distributed system, including (wired and wireless) communication systems, real-time operating systems, embedded systems, automated process control systems, and even biological systems. The behavior of these systems generally includes both discrete state changes and continuous state evolution; Tempo is designed to express both kinds of changes.

The Tempo Toolkit was developed by VEROMODO Inc., with support provided by an AFOSR technology transfer grant. The beta releases of the Tempo Toolkit for Linux, Windows, and Mac OS X platforms are available for download at www.veromodo.com.

Earlier work on a toolkit supporting specification in (untimed) Input/Output Automata was performed at the MIT Theory of Distributed Systems group [9]. The prototype toolkit supported a simulator [6], paired automata simulation [28], and simulations of composed automata [29].

2.2 Tempo language overview

We now discuss the Timed I/O Automata formalism that is the basis of the Tempo language, and summarize the capabilities of the toolkit.

2.2.1 Timed I/O Automata

The Timed I/O Automata [13] mathematical framework is an extension of the classical I/O Automata framework [20, 16], which for many years has been successfully used in the theoretical distributed computing research community to specify and reason about distributed and concurrent algorithms. I/O Automata are very simple interacting asynchronous state machines, without any support for describing timing features. Although they are simple, I/O Automata provide a rich set of capabilities for modeling and analyzing distributed algorithms. I/O Automata support description of many properties that distributed algorithms are required to satisfy, and mathematical proofs that the algorithms in fact satisfy their required properties. These proofs are based on methods such as invariant assertions and compositional reasoning. I/O Automata also support representation of algorithms at different levels of abstraction, and proofs of consistency relationships between

algorithm representations at different levels. Because of these capabilities, I/O Automata have been used fairly extensively for modeling and analyzing asynchronous distributed algorithms, and even for proving impossibility results about computability in asynchronous distributed settings.

However, ordinary I/O Automata cannot be used to describe distributed algorithms that use time explicitly, for example, those that use timeouts or schedule events periodically. And they do not provide explicit support for describing timing constraints such as bounds on message delay or clock rates. Moreover, without support for timing, I/O Automata could not be used for other applications such as practical communication protocols. These limitations led to the development of Timed I/O Automata, which include new features—most notably, *trajectories*—specifically designed for describing timing aspects of systems.

Like ordinary I/O Automata, Timed I/O Automata are simple interacting state machines and have a well-developed, elegant theory, presented in [13]. Like I/O Automata, Timed I/O Automata provide a rich set of capabilities for system modeling and analysis. Methods used for analyzing timed I/O automata are essentially the same as those used for ordinary I/O automata: invariant assertions, compositional reasoning, and correspondences between levels of abstraction.

2.2.2 The Tempo language and tools

I/O Automata and Timed I/O Automata are fine mathematical modeling frameworks for distributed systems and have been used, by hand, to describe and analyze distributed algorithms, communication protocols, and embedded systems. Yet, computer support could make these tasks quite a bit easier. The Tempo Language and Toolkit is an attempt at providing a broad set of tools to support these activities.

The Tempo toolkit contains tools to support analysis of systems. These include a compiler that checks syntax and perform static semantic analysis; a simulator to produce and explore execution traces for an automaton; a translation module to the UPPAAL model-checker [14]; and a translation module to the PVS interactive theorem-prover [27]. The overall architecture of the Tempo toolkit has been designed to facilitate incorporation of other validation tools in the future.

The Tempo language has a rather minimal syntax, which closely matches the simple semantics of the Timed I/O Automata mathematical framework. In fact, the mapping between a Tempo automaton description and the Timed I/O Automata that it denotes is pretty transparent. For example, an automaton's discrete transitions and continuous evolutions are described directly in Tempo, by "transitions" and "trajectories", respectively. The minimality of the Tempo language does not limit its expressive power: Tempo is capable of describing very general systems of Timed I/O Automata. Of course, many analysis tools—especially automated ones like model-checkers—are not capable of handling fully general Tempo programs. In contrast with the conventional approach taken by developers of automated tools, Tempo does not outright limit the expressive power of the language and opts instead for the definition of *sublanguages* that are suitable for use with particular tools.

2.3 An Example: mutual exclusion algorithm

To illustrate the capabilities of Tempo and its simulator, we will be using the Fischer Timed Mutual Exclusion Algorithm. It has become famous as a standard test example for formal methods for modeling and analyzing timed systems. An informal description of the example appears in [16], Chapter 24.

2.3.1 The Tempo specification

This example illustrates most of the basic constructs needed for writing a Tempo program for a single Timed I/O Automaton modeling a shared-memory system. The example also demonstrates how to express invariants using Tempo, including invariants that involve time.

The Tempo model shown in Code 1 and 2 describes the entire system as a single Timed I/O Automaton. The vocabulary section declares the data types used in the algorithm, namely, the abstract data type *process* and the program counter abstract data type *PcValue* (an enumerated type) to represent the exact location of each process in its program. Each process could be in its *remainder region* (program counter = *pc_rem*), where it is not engaged in trying to enter the critical region. Or, it could be about to test, set, or check the *turn* variable. Or, it could be in various stages of entering or leaving the critical region—the model uses separate program counter values to represent situations where the process has successfully completed the trying protocol, where it is actually in the critical region, where it is about to reset the *turn* variable upon leaving, and where it has successfully completed the exit protocol.

The actual automaton description begins with the name of the automaton, with formal parameters *Lcheck* and *u_set*. These are real numbers representing, respectively, a lower bound on the time between setting and checking, and an upper bound on the time between checking and setting. The where clause specifies restrictions imposed on the parameters saying (most importantly) that *u_set* must be strictly less than *Lcheck*. The automaton imports the vocabulary to make its definition available to the remainder of the specification.

The automaton's signature, describe its actions. Actions are classified as **input**, **output**, or **internal**. Here, no input actions are used, i.e., the system is "closed". Since the entire system is being modelled by a single automaton, each type of action is parameterized by the name of the process that performs it. In this model, the internal actions are associated with shared-variable accesses—the steps that test, set, check, and reset the *turn* variable. The output actions are those that mark processes' progress through the various high-level regions of their code: The *try*(i) action describes process *i* moving from its remainder region to its *trying region*, in which it executes a protocol to try to reach the critical region. The *crit*(i) action describes passage from the trying region to the critical region, and the *exit*(*i*) action describes passage from the critical region to the *exit region*, where process *i* performs its exit protocol. Finally, the *rem*(*i*) action describes passage from the exit region back to the remainder region.

The automaton's state is specified in the states section. The shared variable *turn* has type Null[*process*], which indicates that its value can either be a process or the special value *nil* to indicate the absence of value. *turn* is initially set to *nil*. The variable *pc* represents the program counters for all of the processes in an array of *PcValue* indexed by processes. Initially, all of the program counter values are set to *pc_rem*, which means that all of the processes start out in the remainder region.

The remaining three variables are introduced solely to express the needed timing constraints. First, the variable *now* is used to represent the real time. It is initialized to 0.

Second, the variable *last_set* is an array containing absolute real time upper bounds (*deadlines*) for the processes to perform set actions. A deadline will be in force for a process *i* only when its program counter is equal to *pc_set*, that is, when it is in fact ready to set the *turn* variable. In this case, the value of *last_set*[*i*] will be a nonnegative real number; otherwise, that is, if the program counter is anything other than *pc_set*, the value will be ∞, representing the absence of any such deadline. The elements of the *last_set* array are defined to be of type *AugmentedReal*: a type that

```
vocabulary fischer_types
  types process,
  PcValue : Enumeration [pc_rem, pc_test, pc_set, pc_check,
            pc_leavetry, pc_crit, pc_reset, pc_leaveexit]
end

automaton fischer(L_check, u_set: Real)
  where u_set < L_check ∧u_set ≥0 ∧L_check ≥0
  imports fischer_types

  signature
    output try(i: process)
    output crit(i: process)
    output exit(i: process)
    output rem(i: process)
    internal test(i: process)
    internal set(i: process)
    internal check(i: process)
    internal reset(i: process)

  states
    turn: Null![process] : = nil;
    pc: Array[process, PcValue] : = constant(pc_rem);
    now: Real : = 0;
    last_set: Array[process, AugmentedReal] : = constant(∞);
    first_check: Array[process, DiscreteReal] : = constant(0);

  transitions
    output try(i)
      pre pc[i] =pc_rem;
      eff pc[i] : = pc_test;
    internal test(i)
      pre pc[i] =pc_test;
      eff if turn =nil then
            pc[i] : = pc_set;
            last_set[i] : = (now + u_set);
          fi;
    internal set(i)
      pre pc[i] =pc_set;
      eff turn : = embed(i);
          pc[i] : = pc_check;
          last_set[i] : = ∞;
          first_check[i] : = now + L_check;
```

Code 1: Tempo spec. of the Fischer algorithm (I)

```
  internal check(i)
    pre pc[i] =pc_check ∧first_check[i] ≤now;
    eff if turn =embed(i) then
          pc[i] : = pc_leavetry;
        else
          pc[i] : = pc_test;
        fi;
        first_check[i] : = 0;
  output crit(i)
    pre pc[i] =pc_leavetry;
    eff pc[i] : = pc_crit;
  output exit(i)
    pre pc[i] =pc_crit;
    eff pc[i] : = pc_reset;
  internal reset(i)
    pre pc[i] =pc_reset;
    eff pc[i] : = pc_leaveexit;
        turn : = nil;
  output rem(i)
    pre pc[i] =pc_leaveexit;
    eff pc[i] : = pc_rem;
trajectories
  trajdef traj
    stop when
      ∃i: process (now =last_set[i]);
    evolve
      d(now) =1;
```

Code 2: Tempo spec. of the Fischer algorithm (I.)

includes all (positive and negative) real numbers, plus two values corresponding to positive and negative infinity. Initially, since none of the program counters is *pc_set*, the values in the array are all ∞.

Third and finally, the variable *first_check* is an array containing absolute real time lower bounds (*earliest times*) for the processes to perform *check* actions, when their program counters are equal to *pc_check*. The elements of *first_check* are of type *DiscreteReal*, which means that they always have *Real* values, and moreover, they do not change between discrete actions.

The detailed description of the transitions of the automaton follows in the transitions section. Transitions are (state, action, state) triples. The transitions are described in *guarded command* style, using small pieces of code called *transition definitions*. Each transition definition denotes a collection of transitions, all of which share a common action name.

Each transition has a name, list of parameters, a *precondition* that indicates when the action is enabled and finally, an effect clause that describes the changes to the state when that accompany the action. Input actions are always enabled, reflecting the assumption that Timed I/O Automata are *input-enabled*. Notionally, input actions have no preconditions, as a shorthand for the precondition being true.

The *try*(i) transition represents an entrance by process i into its trying region. The transition is allowed to occur whenever *pc*[i] =*pc_rem*, that is, whenever process i is in its remainder region. The effect is simply to advance the program counter to *pc_test* to indicate that process i is ready to test the *turn* variable.

The *test*(i) transition represents process i testing the *turn* variable. It is allowed to occur whenever *pc*[i] =*pc_test*. The transition can either find the *turn* variable equal to *nil* at which point it moves to take the turn (by setting the program counter to *pc_set*) and saves in *last_set*[i] the deadline for the set action to occur at the latest in *u_set* time steps in the future (away from *now*). The transition can also find that *turn* is not *nil* and simply takes no action to remain in the state, ready to test again.

The set(i) transition represents process i setting the *turn* variable to its own index. This is allowed to occur whenever *pc*[i] =*pc_set*. The effects are given as straight-line code in which process i simply sets *turn* to its own index (the *embed* call is necessary to store the value into an object of type Null[*process*]). The code then sets the program counter to *pc_check* to enable the *check*(i) transition that will verify the *turn* variable. Now that the set(i) action has occurred, the *last_set*[i] deadline is reset to its default value, ∞. The code also records the earliest time when process i could recheck the *turn* variable based on the current clock *now* and the lower bound *l_check*.

The *check*(i) transition is enabled when process i's program counter is set to *pc_check* and its earliest checking time has passed (*first_check*[i] $\leq$*now*). When the transition executes, two interesting cases may arise: If process i finds that *turn* is still equal to i, it leaves the trying region and enters the critical region. On the other hand, if it finds the *turn* variable equal to anything else, it gives up the current attempt and goes back to the testing step. In either case, *first_check*[i] is reset to its default, 0.

The subsequent transitions are quite straightforward. A *crit*(i) transition represents process i moving into the critical region, and an *exit*(i) transition represents process i leaving the critical region. A *reset*(i) transition represents process i resetting the *turn* variable to its default value *nil*, and a *rem*(i) transition represents process i returning to its remainder region.

The final part of the automaton description is the set of **trajectories**, that is, the functions from time to states that describe how the state is permitted to evolve between discrete steps. This model

specifies one trajectory definition, named *traj*. This definition describes the evolution of the state in a way that allowed the current time *now* to increase at rate 1. All of the other state variables are of types that are defined to be discrete; these, by default, are not allowed to change during trajectories. The stop when condition says that a trajectory must stop if the state ever reaches a point where the current time *now* is equal to a specified deadline *last_set*[*i*], for any *i*. That is, time is not "allowed to pass" beyond any deadline currently in force.

This stop when condition is an example of a phenomenon whereby an automaton can prevent the passage of time. This may look strange (at first) to some programmers, since programs of course cannot prevent time from passing. However, appearances can be deceiving and the Fischer automaton is not exactly a program; it is a *descriptive model* that expresses both the usual sort of behavior expressed by a program, plus additional timing assumptions that might be expressed in other ways.

2.3.2 Properties of the algorithm

Tempo can be used to describe not just algorithms, but also properties that we would like the algorithms to satisfy. For example, the Fischer algorithm is supposed to satisfy the *mutual exclusion* property, saying that no two processes can simultaneously reside in their critical regions. This is a claim that the mutual exclusion is an *invariant* of the Fischer algorithm, that is, that it is true in all reachable states of the *fischer* automaton. This claim can be expressed in Tempo with a block

invariant of *fischer*:
$\forall i$: *process* $\forall j$: *process*
$(i \neq j \Rightarrow (pc[i] \neq pc\_crit \vee pc[j] \neq pc\_crit))$;

This invariant definition claims that, in any reachable state of the automaton, any two processes cannot simultaneously be in the critical section. This formal statement must, of course, be verified with a tool in order to formally prove that the algorithm is correct. For instance, one could use an interactive theorem prover such as PVS, a model-checker like UPPAAL, or run simulations of the protocol and require the simulator to check the assertions after every single step of the simulations.

In the next sections we describe the Tempo language and integrated development framework, and their design in more detail, and we describe the work carried out in Phase II.

3 Extensions of I/O Automata and Timed I/O Automata Frameworks

We have continued our work on mathematical foundations for modeling and analyzing timed, hybrid, and probabilistic systems. We have been pursuing an effort to extend Timed I/O Automata to allow probabilistic behavior even before the start of Phase I work, resulting in several papers, e.g., [23, 25, 19]. In an extended series of case studies, we have also been using probabilistic (PIOA) and timed I/O automata (TIOA) to model and verify security protocols. This has entailed extending the formal foundations in several directions, to restrict possibilities for nondeterminism, to define appropriate implementation relationships for the security setting, and to integrate timing into security models.

3.1 Probabilistic extensions

Our recent work investigated the time-bounded task-PIOA modeling framework, an extension of the probabilistic input/output automata (PIOA) framework that can be used for modeling and verifying security protocols. Time-bounded task-PIOAs can describe probabilistic and nondeterministic behavior, as well as timebounded computation. Together, these features support modeling of important aspects of security protocols, including secrecy requirements and limitations on the computational power of adversarial parties. They also support security protocol verification using methods that are compatible with less formal approaches used in the computational cryptography research community. We illustrate the use of our framework by outlining a proof of functional correctness and security properties for a well-known oblivious transfer protocol. These results appeared in print in 2008 [4].

We also introduced the notion of approximate implementations for Probabilistic I/O Automata (PIOA) and developed methods for proving such relationships [24]. We employ a task structure on the locally controlled actions and a task scheduler to resolve nondeterminism. The interaction between a scheduler and an automaton gives rise to a trace distribution– a probability distribution over the set of traces. We define a PIOA to be a (discounted) approximate implementation of another PIOA if the set of trace distributions produced by the first is close to that of the latter, where closeness is measured by the (resp. discounted) uniform metric over trace distributions. We propose simulation functions for proving approximate implementations corresponding to each of the above types of approximate implementation relations. Since our notion of similarity of traces is based on a metric on trace distributions, we do not require the state spaces nor the space of external actions of the automata to be metric spaces. We discuss applications of approximate implementations to verification of probabilistic safety and termination.

3.2 Extensions for reasoning about security protocols

In another recent development, we investigated a new paradigm for the analysis of long-lived security protocols. We allow entities to be active for a potentially unbounded amount of real time, provided they perform only a polynomial amount of work per unit real time. Moreover, the space used by these entities is allocated dynamically and must be polynomially bounded. We proposed a key notion of long-term implementation, which is an adaptation of computational indistinguishability to the long-lived setting. We show that long-term implementation is preserved under polynomial parallel composition and exponential sequential composition. To illustrate the use of this new paradigm, we analyze the long-lived timestamping protocol of Haber and Kamat. This work was submitted for publication in 2008 [5]

3.3 Extensions for hybrid systems

We completed our work on a journal paper on average dwell time for hybrid systems [15]. Average dwell time (ADT) properties characterize the rate at which a hybrid system performs mode switches. In this paper, we present a set of techniques for verifying ADT properties. The stability of a hybrid system A can be verified by combining these techniques with standard methods for checking stability of the individual modes of A. We introduce a new type of simulation relation for hybrid automata switching simulation for establishing that a given automaton A switches more rapidly than another automaton B. We show that the question of whether a given hybrid automaton has ADT a can be answered either by checking an invariant or by solving an optimization problem. For

classes of hybrid automata for which invariants can be checked automatically, the invariant-based method yields an automatic method for verifying ADT; for automata that are outside this class, the invariant has to be checked using inductive techniques. The optimization-based method is automatic and is applicable to a restricted class of initialized hybrid automata. A solution of the optimization problem either gives a counterexample execution that violates the ADT property, or it confirms that the automaton indeed satisfies the property. The optimization and the invariant-based methods can be used in combination to find the unknown ADT of a given hybrid automaton.

We developed a new abstraction technique, event order abstraction (EOA), for parametric safety verification of real-time systems in which "correct orderings of events" needed for system correctness are maintained by timing constraints on the systems' behavior [32]. By using EOA, one can separate the task of verifying a real-time system into two parts: 1. Safety property verification of the system given that only correct event orderings occur; and 2. Derivation of timing parameter constraints for correct orderings of events in the system. The user first identifies a candidate set of bad event orders. Then, by using ordinary untimed model-checking, the user examines whether a discretized system model in which all timing constraints are abstracted away satisfies a desirable safety property under the assumption that the identified bad event orders occur in no system execution. The user uses counterexamples obtained from the model checker to identify additional bad event orders, and repeats the process until the model-checking succeeds. In this step, the user obtains a sufficient set of bad event orders that must be excluded by timing synthesis for system correctness. Next, the algorithm presented in the paper automatically derives a set of timing parameter constraints under which the system does not exhibit the identified bad event orderings. From this step combined with the untimed model-checking step, the user obtains a sufficient set of timing parameter constraints under which the system executes correctly with respect to a given safety property. In our documented work we illustrated the use of EOA with a train-gate example inspired by the general railroad crossing problem. We also summarized three other case studies, a biphase mark protocol, the IEEE 1394 root contention protocol, and the Fischer mutual exclusion algorithm.

4 Tempo Toolkit: Architecture and Language

The Phase II STTR [18] completed in 2007 produces a solid implementation of the TIOA language in the form of a toolkit: TEMPO. VEROMODO focused on a redesign of the core implementation of the front-end (analyzer and compiler) and a design of its interfaces to the various back-ends. TEMPO has the following characteristics

- It is a Java 1.5 implementation of a refined TIOA language.
- It offers a modular design to facilitate the integration of additional tools as independent back-ends. (e.g., the PVS translator, the simulator or the model-checker). It is based on modern modular architecture where each back-end tool is a plug-in that can be loaded at runtime to extend the compiler.
- It features a fine-grained interface to communicate with back-end tools that would make it possible to establish a one-to-one correspondence between each back-end tool and the TIOA abstractions offered by TEMPO.

- It offers a flexible integration with the back-ends that let each back-end independently refine the semantic rules to either augment the core language with back-end specific language extensions.

4.1 The Architecture of Tempo

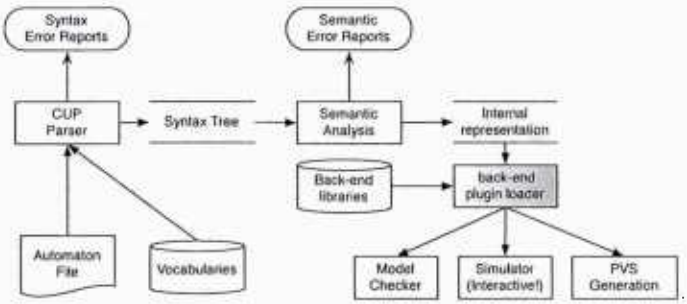

Figure 1: TEMPO's architecture

TEMPO's architecture was designed and developed by our academic partner Laurent Michel (University of Connecticut). and it is based on a modular multi-stage compiler. The overall organization is shown in Figure 1. The first two compiler stages are fixed and independent of the selected back-end tool. The third stage depends upon the selected tool and is loaded automatically from a JAVA shared library (JAR file) based on the user selection at the command line or in the user interface.

The initial stage is responsible for the lexical and syntactic analysis of a TEMPO specification. It assembles its input from one or more text files containing the specifications as well as one or more *vocabularies*. A vocabulary is a TEMPO specification containing built-in abstract data types for commonly used data structures such as sets, multi-sets, maps or arrays[1]. Lexical and grammatical errors are reported immediately. The parser is written with a state-of-the-art freely available parser generator: ANTLR v2.7.x[2]. The output of this phase is an abstract syntax tree that is passed down to a second analysis stage.

The second stage focuses on the semantic analysis of the specification. This phase performs multiple *passes* (traversals) of the AST to analyze it.

From a high-level standpoint, the semantic analysis applies a collection of validation rule to each node of the abstract syntax tree. Each rule take the form

$$P(n) \Rightarrow g_1, \cdots, g_k$$

[1] Users can define their own additional vocabularies which are indistinguishable from TEMPO's own built-in vocabularies

[2] ANTLR v2 is available from `http://www.antlr2.org/`

where P is a boolean predicate on the subtree rooted at n that determines whether the rule is applicable and g_1 through g_k are actions that transform, annotate, or possibly tag the tree as semantically unsound. Each back-end can add a set of validation rules that capture additional requirements on the AST to comply with its limitations. For instance, if a back-end cannot operate on expressions containing quantifiers ($\forall$ or $\exists$), it can add a rule

$$\begin{aligned} class(n) &= ASTForall \vee class(n) = ASTExist \\ &\Rightarrow reject(n, \text{"The back-end xyz does not support quantifiers in expressions"}) \end{aligned}$$

that the semantic analysis will apply inductively (alongside all the other rules) to all the nodes of the abstract syntax tree.

4.2 Tempo language

The TIOA formalism and associated theory is defined in the monograph produced and published as a part of this project [13]. We refer the reader to the monograph for the detailed information about the TIOA formalism, and modeling and analysis methodology.

The TIOA language was refined during the implementation of TEMPO to take into account standard user expectations and to produce an implementation as uniform as possible. The TEMPO language itself has a minimal syntax, which closely matches the simple semantics of the Timed I/O Automata mathematical framework. In fact, the mapping between a Tempo automaton description and the timed I/O automaton that it denotes is pretty transparent. For instance, an automaton's discrete transitions and continuous evolutions are described directly in Tempo by "transitions" and "trajectories", respectively. The minimality of the language does not limit its expressive power: Tempo can describe very general systems of timed I/O automata. Of course, each analysis tool brings its own computational limitations, and Tempo accommodates them with the addition of *tool specific* restrictions (captured through the predicate mechanism described above) to define a suitable sublanguage.

5 Deployment Problems

This section reviews the deployment phase that arise when constructing a distributed systems. We discuss our prior work in the area, then present the language extensions needed to convey the key characteristics of deployment instances, and illustrate an application of our framework in specifying a meaningful sample deployment problem.

Our earlier work on deployment of distributed systems was done in the context of an architectural specification framework called the $\mathcal{I}^5$ [2, 1]. $\mathcal{I}^5$ uses five levels of abstraction, called Interface, Implementation, Integration, Instantiation, and Installation, to describe the hardware and software structures of distributed systems. Deployment of software components to hardware nodes takes place at the Installation level using information gathered at higher levels. $\mathcal{I}^5$ does not incorporate specification of component semantics, and we explored the use of the I/O Automata language in [1] to complement the structural specifications in $\mathcal{I}^5$. Specification of systems in $\mathcal{I}^5$ can be done using UML, but it is not supported by an integrated development environment. The deployment optimization was performed using customized techniques based on binary integer programming and genetic algorithms [3]. Our current work on deployment optimization in TEMPO is in part motivated by $\mathcal{I}^5$. By contrast, TEMPO provides an integrated development environment that

incorporates both structural system descriptions and system semantics, and allows for automatic generation of deployment mappings using advanced constraint-programming techniques.

5.1 Augmenting Tempo with deployment annotations

The Tempo deployment annotations, if any, are part of the definition of a composite automaton. The composite automaton is the only portion of a TIOA model which has multiple components, and it is these component parts which potentially could be deployed to different computing nodes.

The simple composite automaton in Figure 2 illustrates the required deployment annotations. (Section 5.3 has a more realistic example using an Eventually Serializable Data Service (ESDS) [8, 7].) Our example composite automaton consists of two types of components, A and B. Automaton A has two output transitions, a send transition that specifies both a message to be sent and the identifier of its destination, and a gossip transition that specifies data to be broadcast. Automaton B has the two matching input transitions. For simplicity, the state and transition details for automata A and B have been omitted. The composite automaton C has two instances of automaton A, called $a1$ and $a2$, and three instances of automaton B, called $b1$, $b2$, and $b3$.

```
automaton A
  signature
    output send(m : String, id : Nat)
    output gossip(data : Array[Nat, Nat])
  states
  transitions
    output send(m, id)
    output gossip(data)

automaton B(id : Nat)
  signature
    input send(m : String, const id)
    input gossip(data : Array[Nat, Nat])
  states
  transitions
    input send(m, id)
    input gossip(data)

automaton C
  components
    a1 : A;
    a2 : A;
    b1 : B(1);
    b2 : B(2);
    b3 : B(3);

  deployment
    nodes
      n1;
      n2;
      n3;
      n4;

connections
  {n1, n2};
  {n2, n3, n4};

communication
  a1.gossip -> b1, b2, b3 freq 5;
  a2.send -> b3 freq 10;
  a2.send -> b1 freq 2;
```

Figure 2: Simple composite automaton with deployment annotations

The deployment annotations begin with the keyword deployment and contain, at a minimum, a list of the computing nodes, the physical connections among those nodes, and a description of the communication patterns of the composite automaton's components. The list of computing nodes begins with the keyword nodes and is followed by the list of all the computers in the network, namely $n1$, $n2$, $n3$, and $n4$. In this example, node $n1$ is directly connected to $n2$, and nodes $n2$, $n3$, and $n4$ are directly connected to each other by a common connector. This is denoted by the deployment section beginning with the keyword connections and containing, for each set of directly connected nodes, a list of the individual nodes, separated by commas and enclosed in braces.

The last deployment section in this example begins with the keyword communication and lists the relative frequencies with which each of the transitions of the composite automaton occur. For each transition, the component which generates the transition as an output transition is listed

first, followed by a period, the name of the transition, the symbol ->, and the names of all the components which receive this transition as input. The list of input components are followed by the keyword freq and an expression for the relative frequency of the transition. Each frequency expression is interpreted as the number of times the transition occurs during some time period of unspecified length, where it is assumed that the same time period is used in determining the frequencies for all the transitions. Here, component $a1$ broadcasts its gossip message to components $b1$, $b2$, and $b3$ with a relative frequency of 5 while $a2$ outputs its send message to component $b3$ with a relative frequency of 10.

Figure 3 enhances our example composite automaton with some of the optional deployment constraints. The first constraint, beginning with the keyword support, specifies which components may run on which nodes. In our example, node $n1$ supports components $a1$ and $a2$, node $n2$ supports all the components, and nodes $n3$ and $n4$ support components $b1$, $b2$, and $b3$. If no support section is provided in the deployment annotations, every component may be deployed to every node.

```
automaton C
  components
    a1 : A;
    a2 : A;
    b1 : B;
    b2 : B;
    b3 : B;

  deployment
    nodes
      n1;
      n2;
      n3;
      n4;

  connections
    {n1, n2};
    {n2, n3, n4};

  support
    n1 <- a1, a2;
    n2 <- all;
    n3 <- b1, b2, b3;
    n4 <- b1, b2, b3;

  fixed
    n2 <- b1;

  separated
    {a1, a2};
    {b1, b2, b3};

  together
    {a1, b1};

  communication
    a1.gossip -> b1, b2, b3 freq 5;
    a2.send -> b3 freq 10;
    a2.send -> b1 freq 2;
```

Figure 3: Annotations for deployment constraints

The fixed section lists each component which must be deployed to a particular node. Once again, a statement of the form x <- y means that component y must be assigned to host x.

Reliability and fault-tolerance consideration may require that some groups of components be separated or co-located. For instance, data *replicas* should be hosted on different nodes while tightly coupled modules (a communication channel and its replica) should be co-located for efficiency reasons. The separated and together sections can be used to specify these requirements and define lists of sets of modules. In our example, components $a1$ and $a2$ must be assigned to distinct nodes, components $b1$, $b2$, and $b3$ must be assigned to distinct nodes, and components $a1$ and $b1$ must be assigned to the same node.

Figure 4 illustrates more advanced deployment annotations. The first of these is the constants section, which allows the user to name literals[3] used within the specification. Components often will "pass through" some messages, possibly recording information from the messages in their state. It is convenient to specify these common message frequencies using constants. In our example, $f1$ is declared to have the value 5, and $f2$ is declared to have the value 2. Then $f1$ and $f2$ are used

[3]Currently only of type Nat.

to specify the relative frequencies of the three transitions in the communication section.

automaton C	node types	equivalent
components	pc, sun;	$\{n3, n4\}$;
$a1 : A$;		
$a2 : A$;	nodes	support
$b1 : B$;	$n1 :\ pc$;	$n1$ <- $a1, a2$;
$b2 : B$;	$n2$;	$n2$ <- all;
$b3 : B$;	$n3 :\ sun$;	sun <- $b1, b2, b3$;
	$n4 :\ sun$;	
deployment		communication
constants	connections	$a1$.gossip -> $b1, b2, b3$ freq $f1$;
$f1 :$ Nat := 5;	$\{n1, n2\}$ bandwidth 25;	$a2$.send -> $b3$ freq $f1 + f2$ msgSize 4;
$f2 :$ Nat := 2;	$\{n2, sun\}$;	$a2$.send -> $b1$ freq $f2$ msgSize 8;

Figure 4: More advanced deployment annotations

Node types represent groups of nodes with the same capabilities. Anywhere a group appear in a specification, the node type may be used instead. The node types of a deployment must be declared with the keywords node types followed by comma-separated list of node names and terminated with a semicolon. Our example declares two node types, pc and sun. Node $n1$ is of node type pc, and nodes $n3$ and $n4$ are members of sun while node $n2$ has no node type. Whenever pc appears in the specification, it is replaced with node $n1$, and whenever sun is used, it is replaced with nodes $n3$ and $n4$. For instance, the connection among nodes $n2$, $n3$, and $n4$ may be specified as $\{n2, sun\}$.

Some groups of nodes are completely equivalent, in that they support the same set of components and are connected to other nodes in an equivalent manner. Specifying that these nodes are equivalent enables the opimizer to be more efficient. The sets of equivalent nodes are listed in the equivalent section. In our example, nodes $n3$ and $n4$ are equivalent.

In some applications the amount of data transmitted with each transition is essentially the same, but in other applications the amount of data transmitted varies from one transition to another. The deployment annotations allow the size of the transmitted data to be specified for each transition. The optional stanza msgSize *expr* may be added to each transition listed in the communication section. Each message size expression is interpreted as a multiplicative factor of an unspecified unit of transmitted data. In our example, the gossip messages from component $a1$ to components $b1$, $b2$, and $b3$ are of size 1, the send messages from component $a2$ to component $b3$ are of size 4, and the send messages from component $a2$ to component $b1$ are of size 8.

A connection may have a bandwidth limitation. This is specified by appending the stanza bandwidth *expr* where the expression specifies the maximum bandwidth for the set of nodes in the corresponding connection. The bandwidth expression is interpreted as the maximum amount of data which may pass through the connection during a time period, expressed as a factor of a unit of transmitted data. The implementation assumes that each transition uses a single path for data transmission. In our example, the connection between nodes $n1$ and $n2$ has a maximum bandwidth of 25.

5.2 Language Extensions for Deployment Annotations

Deployment annotations are added to the Tempo language as an optional extension to the definition of a composite automaton. The deployment specification begins with the keyword `deployment`

```
composedAutomaton ::=components hiddenActionSets? compSchedule? deployment?

deployment ::='deployment' constants? nodeTypes? nodes
                           connections equivalent? constraint* communication

constants ::='constants' (constant ; )+
constant ::=ID : typeRef := expr

nodeTypes ::='node' 'types' nodeType (, nodeType )* ;
nodeType ::=ID

nodes ::='nodes' (node ; )+
node ::=ID ([ varList (, varList )* ] )? (: nodeType )? deployWhere?

connections ::='connections' ({ nodeSpecList } ('bandwidth' expr )? ; )+
equivalent ::='equivalent' ({ nodeSpecList } ; )+

constraint ::=support
        | together
        | separated
        | fixed
support ::='support' (nodeSpec < - ('all' | compSpecList ); )+
together ::='together' ({ compSpecList } ; )+
separated ::='separated' ({ compSpecList } ; )+
fixed ::='fixed' (nodeInstance < - compInstance ; )+

communication ::='communication' commSpec +
commSpec ::='for' ID 'in' INT . . INT 'do' commSpec + 'od'
        | commTransition
commTransition ::=compTransition - > compSpecList 'freq' expr ('msgSize' expr )? ;
compTransition ::=compInstance . ID (( expr (, expr)* ) )?

nodeSpecList ::=nodeSpec (, nodeSpec )*
nodeSpec ::=nodeInstance deployWhere?
        | nodeType
nodeInstanec ::=ID ([ expr (, expr)* ] )?

compSpecList ::=compSpec (, compSpec )*
compSpec ::=compInstance deployWhere?
compInstance ::=ID ([ expr (, expr)* ] )?

deployWhere ::='where' paramRange (^ paramRange )*
paramRange ::=ID '\in' INT . . INT

plainOp ::=as before | . .
expr ::=as before | expr (. . expr )+
```

Figure 5: EBNF Grammar fragment for deployment expressions.

followed by optional constants and node types specifications, required nodes and connections specifications, optional equivalent nodes and constraint specifications, and a required communication specification. The components to be deployed to a network are the component parts of the composite automaton.

The constants specification, if present, begins with the keyword `constants` followed by one or more declarations of constant variables. Each declaration begins with an identifier, corresponding to the name of the variable, followed by a colon, the data type of the variable, the assignment operator :=, the value of the variable, and a semicolon. The scope of a constant variable declaration is the body of the deployment specification. As the name implies, the value of a constant variable cannot be changed. Constants may be used in expressions to specify bandwidth limitations of connections and frequencies and message sizes of communicating transitions, for example.

The node types specification, if present, consists of the keywords `node types`, one or more identifiers, separated by commas, and a semicolon. Each identifier is the name of a node type, which is just a shorthand name for a group of nodes. A node may belong to at most one node type.

The mandatory nodes specification identifies the host computer nodes onto which the Tempo components are to be deployed. It begins with the keyword `nodes` followed by one or more node declarations, each ending with a semicolon. Each node declaration begins with an identifier, corresponding to the name of the node, and a list of its parameters, if any, separated by commas and enclosed in square brackets. Each parameter specification consists of an identifier, corresponding to the local name of the parameter, followed by a colon and its data type. If multiple, adjacent parameters are of the same data type, their identifiers may be separated by commas and followed by a single colon and their common data type. After the node name and parameters, there is an optional node type designation, consisting of a colon and the identifier of the node's type, and an optional where clause.

A node's where clause specifies the ranges of values for the node's parameters. It begins with the keyword `where`, followed by one or more parameter range specifications, separated by the AND operator `/\`, and ends with a semicolon. Each identifier used within the node's parameter specifications must have a corresponding parameter range specification in the where clause consisting of the identifier, the keyword `\in`, and the integer lower and upper bounds for the identifier's values, separated by two periods (. .).

The mandatory connection section itemizes the hardware communication links in the network, be they simple communication cables connecting two nodes or Ethernet cables or switches connecting multiple nodes. The section begins with the keyword `connections` and contains, for each link, the list of directly connected nodes, separated by commas, enclosed in braces, and terminated with a semicolon. If a link has limited bandwidth, that is specified, after the closing brace but before the terminating semicolon, with the keyword `bandwidth` followed by a measure of the limited capacity. If no bandwidth is specified for a link, it is assumed that the bandwidth of the connection is sufficient to be considered unlimited for the purposes of deployment.

For each connection, each node specification consists of an identifier, corresponding to the name of the node, and a list of its parameters, if any, separated by commas and enclosed in square brackets. An optional where clause may be used to refer to a group of nodes, where each identifier used within the node's parameter specifications must have a corresponding parameter range specification in the where clause, as above. A node type identifier also may be used to refer to a group of nodes for a connection, if all the nodes of that type are connected with a single communication link.

Equivalent nodes, if any, are listed next, beginning with the keyword `equivalent` followed by

each group of equivalent nodes. Within each group the individual nodes or groups of nodes are specified in the same manner as for connections, with the node specifications separated by commas, enclosed in braces, and terminated with a semicolon. Providing the sets of equivalent nodes enables the optimal deployment to be calculated more efficiently.

Several constraints may be placed on the deployment of components to nodes. For example, some components may execute only on a subset of the network's nodes. Some components must be deployed to the same node, while other components must not be co-located. Finally, some components must be deployed to particular nodes.

Each component specification consists of an identifier, corresponding to the name of the component, and a list of its parameters, if any, separated by commas and enclosed in square brackets. An optional where clause may be used to refer to a group of components, where each unbound identifier used within the component's parameter specifications must have a corresponding parameter range specification in the where clause, as for nodes.

The support constraints, if present, specify which components may be deployed to which nodes. The section begins with the keyword `support` and gives for each node or group of nodes the list of components they support. Each individual support constraint begins with an identifier, corresponding to the name of a node or node type. If the identifier corresponds to the name of a node, it is followed by the list of the node's parameters, if any, separated by commas and enclosed in square brackets, and an optional where clause specifying the range of values for the node's parameters. The node specification is followed by the symbol `<-` and either a list of specifications for the supported components, separated by commas, or the keyword `all` if the nodes support all components. Each support constraint ends in a semicolon. If no support constraints are included in a deployment specification, every component may run on every node; otherwise, a support constraint must be supplied for each node.

The together constraints, if present, specify groups of components that must be deployed together to the same nodes. The section begins with the keyword `together` and consists of groups of component specifications, separated by commas, enclosed in braces, and terminated with semicolons.

Similarly, the separated constraints, if present, specify groups of components that must be deployed to separate, distinct nodes. The section begins with the keyword `separated` and consists of groups of component specifications, separated by commas, enclosed in braces, and terminated with semicolons.

The fixed constraints, if present, identify the components that must be deployed to particular nodes. The section begins with the keyword `fixed`. Each individual fixed constraint begins with an identifier, corresponding to the name of the node onto which the component is to be deployed, and a list of its parameters, if any, separated by commas and enclosed in square brackets. This is followed by the symbol `<-` and a second identifier, corresponding to the name of the component, and a list of its parameters, if any, separated by commas and enclosed in square brackets. Each constraint ends with a semicolon. Since a fixed constraint assigns a single component to a single node, neither a where clause nor a node type may be used in the specifications.

The final deployment section, a mandatory communication section, specifies the frequencies of the composite automaton's communicating transitions. It consists of the keyword `communication` followed by the individual transition specifications. Each transition specification begins with an identifier, corresponding to the name of the "sending" component, a list of its parameters, if any, separated by commas and enclosed in square brackets, followed by the dot symbol `.`, and a second

identifier, corresponding to the name of one of the component's transitions. These are followed by the symbol -> and a list of component specifications, separated by commas, for the "receiving" components. The transition specification ends with the keyword `freq` followed by an expression for the frequency of the transition, optionally the keyword `msgSize` and an expression for the average size of the transition's "message", and a semicolon. The transition must be an output transition of the "sending" component and an input transition of each of the "receiving" components. The units and time interval for the transition frequency and message size expressions are not specified as part of the deployment annotations: it is assumed that application-specific units are selected and uniformly used for all transition specifications and connection bandwidth limitations.

A for loop can be used to specify groups of similar transitions, such as those of gossiping data replicas. The for loop begins with the keyword `for`, an identifier for the loop variable, the keyword `in`, integers for the lower and upper bounds on the loop variable, separated by the symbol .., and the keyword `do`. These are followed by one or more transition specifications, as above, and the keyword `od`. Each occurrence of the loop variable among the parameters of the "sending" and "receiving" component specifications is replaced, in turn, by each value between the loop variable's bounds (inclusively).

5.3 Eventually Serializable Data Service Annotations

An Eventually-Serializable Data Service (ESDS) [8, 7] maintains multiple copies of its data for fault tolerance, but it selectively relaxes the consistency requirements among its copies of the data in exchange for improved performance. ESDS guarantees that the replicated data will eventually be consistent, although it may not be at a particular point during the execution.

ESDS consists of four types of components: clients, front ends, replicas, and channels. The clients request operations to be performed on the shared data and receive responses containing the results of these operations. The front ends communicate with the clients, keeping track of all their pending requests and forwarding those requests to one or more of the replicas. Each replica keeps a copy of the requested operations on the shared data and a partial order on those operations; the partial order must be consistent with both the responses and the eventual total ordering of the operations. The front ends do not send every request to every replica, so the replicas "gossip" among themselves to stay informed about all the operations that have been received and processed. The channels are used to transmit these gossip messages.

Figures 6 and 7 illustrate the component communication of an example ESDS and the computer network onto which it is to be deployed. This example first appeared in [1]. More recently, the example was hand-coded in Comet to test the feasibility of using constraint programming to determine optimal deployments [21]. Figure 8 contains the Tempo deployment annotations for this example.

The example consists of four clients, $c[1]$, $c[2]$, $c[3]$, and $c[4]$, two front ends, $fe[1]$ and $fe[2]$, and six replicas, $r[1]$, $r[2]$, $r[3]$, $r[4]$, $r[5]$, and $r[6]$. Clients $c[1]$ and $c[2]$ make their requests of $fe[1]$, and clients $c[3]$ and $c[4]$ make their requests of $fe[2]$. Front end $fe[1]$, in turn, forwards its requests to $r[1]$, and front end $fe[2]$ forwards its requests to $r[4]$. The components are to be deployed to four PCs, $pc[1]$, $pc[2]$, $pc[3]$, and $pc[4]$, and ten Sun servers, $sun[1]$ through $sun[10]$. Each of the PCs is connected to a Sun, and all of the Suns are connected to each other with a common connector.

Several additional requirements are placed on the deployment. First, $c[1]$, $c[2]$, and $c[3]$ must be deployed to PCs; the rest of the components must be deployed to Suns. Second, to maintain fault tolerance, the replicas must be deployed to distinct computers. Third, $fe[1]$ must be deployed

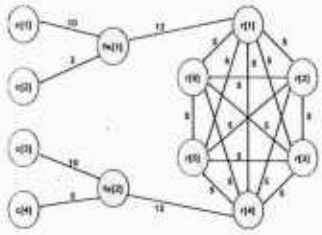

Figure 6: Components for ESDS example.

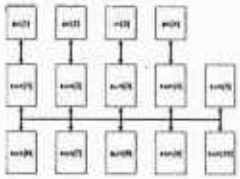

Figure 7: Nodes for ESDS example.

to $sun[2]$, and $fe[2]$ must be deployed to $sun[3]$. This last requirement was added to make the deployment optimization more tractable in its initial implementation [1].

The Tempo implementation of ESDS requires channels between each pair of replicas, making the model more consistent with the original ESDS model [8]. These channels are named $ch[1,1]$ through $ch[6,6]$, where replica $r[i]$ uses channel $ch[i,j]$ to gossip with replica $r[j]$. The channels require an additional set of deployment constraints, namely, each replica $r[i]$ must be co-located with each of its channels $ch[i,j]$.

The *ESDS* automaton in Figure 8 stores both its components and its nodes in arrays. For example, the Client components are declared in the components section with

$$c[i : \mathsf{Nat}] : \mathit{Client}(i) \text{ where } i \text{ \in } 1..4;$$

Note that the data type of the array index must be declared as an Nat. The range of the array indices is specified with the keyword where followed by the index variable, the keyword \in, the lower bound of the indices, two periods, and the upper bound of the indices. Array indices need not start with 1. Both the component and node arrays may be multi-dimensional, such as the array *ch* of Channel components.

The components and nodes that are stored in arrays may be accessed both individually, such as $pc[3]$, or as a group of sequential elements, such as $sun[i]$ where i \in 5..10 in the equivalent section. Again, a where clause is used to specify the range of array indices. Note that the range of indices may be used to specify a subset of the elements in an array.

In the communications section nested for loops may be used to specify similar transitions for arrays of components. This is particularly helpful in the *ESDS* automaton for specifying the gossip frequencies of the 72 transitions among the replicas and channels.

6 Generating deployment models

The deployment annotations are incorporated into the Tempo Toolkit [17] as a new plug-in. The plug-in translates the annotations into a Comet constraint program, which is subsequently executed to determine an optimal allocation of components to computing nodes in the target network. We now describe in detail the translation scheme, the resulting Comet program, and the language restrictions designed to enable effective automatic generation of optimal deployment.

```
automaton ESDS
  components
    c[i : Nat] : Client(i) where i \in 1..4;
    fe[i : Nat] : FrontEnd(i) where i \in 1..2;
    r[i : Nat] : Replica(i) where i \in 1..6;
    c[i : Nat, j : Nat] : Channel(i, j) where
        i \in 1..6 /\ j \in 1..6;
  deployment
    constants
      c1Freq : Nat := 10;
      c2Freq : Nat := 2;
      c3Freq : Nat := 10;
      c4Freq : Nat := 5;
      gossipFreq : Nat := 5;
    nodes
      pc[i : Nat] where i \in 1..4;
      sun[i : Nat] where i \in 1..10;
    connections
      {pc[1], sun[1]};
      {pc[2], sun[2]};
      {pc[3], sun[3]};
      {pc[4], sun[4]};
      {sun[i] where i \in 1..10};
    equivalent
      {sun[i] where i \in 1..4};
      {sun[i] where i \in 5..10};
    support
      pc[i] where i \in 1..4 <- c[1], c[2], c[3];
      sun[i] where i \in 1..10 <- c[4], fe[1], fe[2],
          r[i] where i \in 1..6;
      sun[i] where i \in 1..10 <- c[i, j] where
          i \in 1..6 /\ j \in 1..6;
  fixed
    sun[2] <- fe[1];
    sun[3] <- fe[2];
  separated
    {r[i] where i \in 1..6};
  together
    {r[1], ch[1, j] where j \in 1..6};
    {r[2], ch[2, j] where j \in 1..6};
    {r[3], ch[3, j] where j \in 1..6};
    {r[4], ch[4, j] where j \in 1..6};
    {r[5], ch[5, j] where j \in 1..6};
    {r[6], ch[6, j] where j \in 1..6};

  communication
    c[1].request -> fe[1] freq c1Freq;
    c[2].request -> fe[1] freq c2Freq;
    c[3].request -> fe[2] freq c3Freq;
    c[4].request -> fe[2] freq c4Freq;
    fe[1].send -> r[1] freq c1Freq + c2Freq;
    fe[1].response -> c[1] freq c1Freq;
    fe[1].response -> c[2] freq c2Freq;
    fe[2].send -> r[4] freq c3Freq + c4Freq;
    fe[2].response -> c[3] freq c3Freq;
    fe[2].response -> c[4] freq c4Freq;
    r[1].receive -> fe[1] freq c1Freq + c2Freq;
    r[4].receive -> fe[2] freq c3Freq + c4Freq;
    for i in 1..6 do
      for j in 1..6 do
        r[i].gossipSend -> ch[i, j] freq gossipFreq;
        ch[i, j].gossipReceive -> r[j] freq gossipFreq;
      od
    od
```

Figure 8: Deployment annotations for ESDS example

6.1 Translation Scheme

The deployment annotations are extensions to the Tempo language, so care was taken to minimize their impact on existing Tempo programs. To that end, the deployment annotations only occur within the definition of composite automata, and they are isolated within those definitions to a separate new section beginning with the keyword deployment.

The translation process begins by enumerating all the components and nodes and assigning their names, as provided by the Tempo modeler, to two arrays of type string. The Comet program then identifies the components and nodes by the indices of their names in these arrays. For example, for the deployment annotations in Figure 2 the array of node names is $["n1","n2","n3","n4"]$ and the connection sets $\{n1,n2\}$ and $\{n2,n3,n4\}$ are encoded as $\{0,1\}$ and $\{1,2,3\}$. At the end of execution, the Comet program displays the optimal deployment with the Tempo modeler's names. Figure 9 shows the resulting deployment output for all but the channel components of the ESDS example in Section 5.3.

```
Deployment:
pc[2] <- c[1]
pc[2] <- c[2]
pc[3] <- c[3]
sun[3] <- c[4]
sun[2] <- fe[5]
sun[3] <- fe[6]
sun[2] <- r[7]
sun[5] <- r[8]
sun[1] <- r[9]
sun[3] <- r[10]
sun[4] <- r[11]
sun[6] <- r[12]
  ⋮
```

Figure 9: Deployment for ESDS Example

The variables declared in a constants section of the deployment specification are carried over to the Comet program and declared and initialized there. When these variables are subsequently used to specify communication frequencies, for example, the variable names, rather than their values, are encoded in the Comet program. This allows arbitrary arithmetic expressions for communication frequencies without requiring the Tempo front-end to evaluate those expressions.

6.2 Comet Program

The output of the translation stage is a COMET program. That program relies on Constraint Programming technology to solve the deployment problem optimally. Constraint programming delivers a complete solution method. Constraint programs revolve around two components. A declarative component state the discrete decision variables, the constraints that every solutions must satisfy and the objective function. The second component focuses on the specification of a tree-search process revolving around an implicit enumeration.

The TEMPO translator for COMET produces a complete model that features both the declarative component and an instantiation of a search template. That template takes advantage of the properties conveyed through the annotations such as the equivalence classes (specified in the equivalent section) among nodes to implement a symmetry breaking procedure thatconsiderably reduces the running time.

As with equivalent and support, the generated COMET code varies depending upon whether or not bandwidth constraints are included in the deployment specification. Five different interpretations of the bandwidth constraints were considered.

- A single path is used between each pair of nodes.
- A single path is used between each pair of components.
- A single path is used for each transition.

- A single path is used for each message.
- Multiple paths may be used for a single message.

We chose to use a single path between each pair of components since that option most closely embodies the concept of establishing a connection between components. Subsequent versions of the deployment plug-in may include other types of bandwidth constraints, or even include communication load balancing among the connections.

Performance-wise, the programs generated with the help of the TEMPO translator are more than competitive with hand-written programs. When applied to the ESDS deployment, the generated program is, to this date, the most effective way to solve the problem. The effectiveness of this approach, when compared with modern mixed-integer programming solvers, is reported in [22]. For the ESDS example in Section 5.3, the COMET program finds the optimal deployment approximately 20 times faster than CPLEX version 11 and 25,000 times faster than the hand-coded C program reported in [1].

6.3 Tempo Language Restrictions

Each of the Tempo Toolkit plug-ins place some restrictions on the Tempo language constructs which are supported, and the deployment plug-in is no exception. First, since the components and nodes must be enumerable, the contents of their where clauses currently are limited to range sets of type Nat and the /\ operator, as in c[i,j] where i \in 1..6 /\ j \in 1..6. Second, nested composite automata are not supported, pending identification of distributed systems that require this modeling complexity.

Tempo specifies the communication among components implicitly; each output transition is linked with all input transitions having the same name and matching parameters in other components. One of an output transition's parameters often specifies an identifier for the component with the matching input transition. This is particularly useful for applications using arrays of components.

Unfortunately, this implicit linking through parameter values makes it extremely difficult for the Tempo deployment annotations to match output transitions with input transitions at compile-time as needed, rather than run-time. The current annotations use explicit, rather than implicit, transition matching as a result. For example from Figure 2, $a2$.send -> $b3$ freq 10; gives the frequency of the send output transition of $a2$ when it is linked with the send input transition of $b3$, and $a1$.gossip -> $b1, b2, b3$ freq 5; gives the frequency of the gossip output transition of $a1$ when it is linked with the gossip input transitions of $b1$, $b2$, and $b3$. The downside of this approach is that the Tempo front end can only do limited error checking. In the first example, the front end ensures that $a2$ and $b3$ have send transitions of the proper type, but it does not ensure that the transitions actually will link in a run-time setting nor does it ensure that there aren't additional send input transitions in other components which also will link with the output transition. An alternate communication syntax being considered is $a2$.send(_,3) freq 10;, which implicitly links the parameter 3 to a parameter of type const in the send input transition of component $b3$.

7 Solving deployment models

This section describes the optimization model that one obtains from the TEMPO translator when it is applied to the Eventually Serializable Data Service application. The section starts with a pre-

sentation of the abstract model followed by its incarnation in COMET as a constraint programming model.

7.1 The Abstract Model

The input data consists of

- The set of software modules C.
- The set of hosts N.
- For each component, the subset of hosts to which it can be assigned. In the following, $s_{c,n}$ is a boolean variable equal to *true* if and only if component c can be assigned to host n.
- The network cost is directly derived from its topology and expressed with a matrix h where $h_{i,j}$ is the minimum number of hops required to send a message from host i to host j. Note that $h_{i,i} = 0$ (local messages are free).
- The message volumes. In the following, $f_{a,b}$ denotes the average frequency of messages sent from component a to component b.
- The separation set *Sep* which specifies that the components in each $S \in Sep$ must be hosted on a different servers;
- The co-location set *Col* which specifies that the components in each $S \in Col$ must be hosted on the same servers;

The decision variables x_c are associated with each module $c \in C$ and $x_c = n$ if component c is deployed on host n. An optimal deployment minimizes

$$\sum_{a \in C} \sum_{b \in C} f_{a,b} \cdot h_{x_a, x_b}$$

subject to the following constraints. Each component may only be assigned to a host that supports it

$$\forall c \in C \ : \ x_c \in \{i \in N \mid s_{c,i} = 1\}.$$

For each separation constraint $S \in Sep$, we impose

$$\forall i, j \in S \ : \ i \neq j \Rightarrow x_i \neq x_j.$$

Finally, for each co-location constraint expressed over a subset of components $S \in Col$, we impose

$$\forall i, j \in S \ : \ x_i = x_j.$$

7.2 The CP Model

The COMET constraint program generated by TEMPO for the Eventually Serializable Data Service Deployment Problem is shown in Figure 10. We review its main components.

```
1 range Caps = 1..nbCap;
2 range Colors = 1..nbColors;
3 range Orders = 1..nbOrders;
4 range Slabs = 1..nbSlabs;
5 int capacities[Caps] = ...;
6 int weight[Orders] = ...;
7 int color[Orders] =...;
8
9 set{int} colorOrders[c in Colors] = filter(o in Orders) (color[o] == c);
10
11 int maxCap = max(i in Caps) capacities[i];
12 int loss[c in 0..maxCap] = min(i in Caps: capacities[i] >= c) capacities[i] - c;
13
14 Solver<CP> m();
15 var<CP>{int} x[Orders](m,Slabs);
16 var<CP>{int} l[Slabs](m,0..maxCap);
17
18 minimize<m> sum(s in Slabs) loss[l[s]]
19 subject to {
20    m.post(multiknapsack(x,weight,l));
21    forall(s in Slabs)
22       m.post(sum(c in Colors) (or(o in colorOrders[c]) (x[o] == s)) <= 2);
23 } using {
24    forall(o in Orders) by (x[o].getSize(),-weight[o]) {
25       int ms = max(0,maxBound(x));
26       tryall<m>(s in Slabs: s <= ms + 1)
27          m.label(x[o],s);
28       onFailure
29          m.diff(x[o],s);
30    }
31 }
```

Figure 10: The Constraint-Programming Model in COMET

7.2.1 The Model

The model is depicted in lines 1–21 in Figure 10. The data declarations are specified in lines 2–10 and should be self-explanatory. The decision variables are declared in line 10 (they are the same as in the ESDS model given earlier): variable $x[c]$ specifies the host of component c and its domain is computed from the support matrix s.

The objective function is specified in lines 12-13 and eliminates the diagonal elements (since $h_{i,i} = 0$ for every $i \in N$). The CP formulation features a two-dimensional *element* constraint since the matrix h is indexed by variables. Lines 15-18 state the co-location constraints: for each set S (line 15), an element $c_1 \in S$ is selected (randomly) and the model imposes the constraint $x_{c_1} = x_{c_2}$ for each other elements c_2 in S. Lines 19-20 state the separation constraints for every set in Sep using alldifferent constraints. The `onDomains` annotations indicate that arc-consistency must be enforced on the equations and alldifferent constraints.

It is interesting to discuss the pruning performed by the objective function when an upper bound is available. In COMET, the multi-dimensional *element* constraints are implemented in terms of a

table T which contains all the tuples

$$\langle a, b, h_{a,b} \rangle \quad (a, b \in C).$$

COMET also creates a new variable $\sigma_{a,b}$ for each term h_{x_a,x_b} in the objective and imposes the constraint

$$(x_a, x_b, \sigma_{a,b}) \in T$$

on which it achieves arc consistency. With this in place, the objective then becomes

$$\sum_{a \in C} \sum_{b \in C} f_{a,b} \cdot \sigma_{a,b}.$$

7.2.2 The Search Procedure

The search procedure is depicted in lines 23–29. It is a variable labeling with dynamic variable and value orderings. Lines 24–28 are iterated until all variables are bound (line 23) and each iteration nondeterministically assigns a variable `x[i]` to a host n (lines 25–26).

It is interesting to review the variable and value orderings which are motivated by the structure of the objective function

$$\sum_{i \in C} \sum_{j \in C} f_{i,j} \cdot h_{x_i,x_j}.$$

In the objective, the largest contributions are induced by assignments of components i and j that are communicating heavily and are placed on distant hosts. As a result, the variable and value ordering are based on two ideas:

1. Assign first a component i whose communication frequency $f[i, j]$ with a component j is maximal (line 24);
2. Try the hosts for component i in increasing number of hops required to communicate with component j (line 25).

The variable selection thus selects first components with the heaviest (single) communications, while the value selection tries to deploy the components to minimize the number of hops.

Arc-Consistency for filtering The CP model used here is quite elegant since it enforces arc consistency on all constraints and the objective function. One may wonder whether arc consistency is critical in ESDSDPs or whether a weaker form of consistency is sufficient. Table 1 depicts a comparison of a bound-consistency model and an arc-consistency model on a collection of synthetic benchmarks. The second and the third column report the results of the CP solver when bound consistency is enforced on the objective, while the fourth and the fifth columns report the performance for the arc-consistency model. The experimental results show a dramatic loss in performance when arc consistency is not used and underline the importance of using sophisticated constraint programming techniques to deliver the desired performances.

Algo	CP-BC		CP-AC	
Bench	T_{end}	$\#CHPT$	T_{end}	$\#CHPT$
SIMPLE2	1.20	7582	0.23	2510
SIMPLE1	6.11	46874	1.38	15408
SIMPLE0	37.21	307365	7.75	87491
fe3c5pc	94.81	748118	2.76	14597
fe3c5pc5	639.87	4705378	4.64	24130
fe3c5sun	166.39	1336353	6.29	30621
fe3c6pc5	1039.20	7238665	3.54	18547
fe3c7pc5	2107.10	14446831	7.83	35726
fe3c7pc5CS	1916.56	12940789	7.77	35312
fe3c7pc5CST	1286.37	8557292	13.68	70495
fe3dist	93.64	839781	4.16	29750
SCSS1SNUFE	62.80	482601	43.34	392628
SCSS2SNUFE	60.47	442373	66.43	380117
SCSS2SNCFE	30.41	246228	50.83	322472
HYPER8	7653.06	33628203	65.07	123213
HYPER16	34570.90	156832040	237.53	513051

Table 1: The Value of Arc Consistency for the CP Model

Exploiting Value Symmetries As discussed earlier, some instances of the ESDS deployment problem feature a variety of symmetries, which can be removed to improve the search performance without sacrificing optimality guarantees. Techniques for removing these symmetries during search are well-known (see, for instance, [33]).

Figure 11 illustrates how to enhance the search procedures presented earlier with symmetry breaking. The sets of equivalent hosts are supplied as additional input data and are used to determine the set of non-equivalent hosts (lines 1–2). Each iteration of the search procedure calculates the set of nodes that are bound in line 6 and the set of nodes that are eligible to host the next component with lines 7 through 11. Line 7 starts by initializing the *searchNodes*, to all the non-equivalent nodes plus all the nodes on which components are already deployed. The loop in lines 8–11 simply adds to *searchNodes* one still unused node from each equivalence class.

8 A Formal Treatment of an Abstract Channel Implementation Using Java Sockets and TCP

Our earlier research substantiates our ability to implement practical techniques for generating distributed code automatically, starting from formal Input/Output Automata (IOA) specifications in Tempo. Namely, we have developed an automated code-generator for IOA programs in a specific node-channel form that produces Java code running over MPI on a local area network [30, 31], and have used this to generate running versions of a variety of basic distributed algorithms [10]. We have also developed two complete distributed systems by manually (but systematically) translating formal IOA specifications to distributed code, using C++/MPI to implement an eventually-serializable data service [7], and using Java/sockets to implement a reconfigurable atomic read/write memory service, called Rambo, e.g., see [26, 11]. The methodology that emerged as a part of the develop-

```
1  set{set{int}} Eq = ...; //The equivalent host sets
2  set{int} NotEq = ...; //The non-equivalent hosts
3
4  while (sum(k in C) x[k].bound() < C.getSize()) {
5     selectMax(a in C: !x[a].bound(), b in C)(f[a,b]) {
6        set{int} boundNodes = collect(s in C : x[s].bound()) x[s];
7        set{int} searchNodes = notEq union(union(e in Eq) e inter boundNodes);
8        forall (e in Eq) {
9           set{int} fen = e \ boundNodes;
10          if (card (fen) > 0) searchNodes.insert(min (n in fen) n);
11       }
12       int k = min(k in N : x[c2].memberOf(k)) k;
13       tryall<m>(n in searchNodes : x[c1].memberOf(n)) by (h[n, k])
14          cp.post(x[a] == n);
15       onFailure
16          cp.post(x[a] != n);
17    }
18 }
```

Figure 11: The Search Procedure with Value Symmetry Breaking

ment of the latter system (Rambo) will be the basis for prototype implementation and eventual production-grade compiler for Tempo.

As a part of this effort, we have addressed the problem of mapping Tempo-specified channels used in dynamic distributed systems to executable code *en route* to prototyping automated code generation.

Abstract models and specifications can be used in the design of distributed applications to formally reason about their safety properties. However, the benefits of using formal methods are often negated by the ad hoc process of mapping the functionality of an abstract specification to the low-level executable code for target distributed platforms. We have developed the first formal specification of an abstract asynchronous communication channel with support for dynamic creation and tear down of communication links between participating network nodes, and its implementation using Java sockets. The specifications are expressed using the Tempo formalism, and it is proved that the resulting implementation preserves the safety properties of the abstract channel. This approach can be used to implement algorithms for dynamic systems, where communicating nodes may join, leave, and experience arbitrary delays. This directly benefits automated code generation we are targetting in this project, and we plan to include an implementation of such channels in the Tempo toolkit as a standard building block for dynamic distributed systems. Our results appear in the proceedings of 2008 *IEEE International Symposium on Network Computing and Applications* [12].

8.1 Rationale: towards code generation

The increasing complexity of distributed software systems makes reasoning about their behavior evermore challenging. Abstract specifications of distributed systems simplify formal reasoning about their safety guarantees, and several formal systems have been used for this purpose. However, this abstraction makes challenging the mapping of the high-level specification to the facilities available in a target programming language.

Translation of abstract specifications into executable code for target environments is particularly challenging in the case of communication channels. Distributed services are designed for a specific communication model, where the safety properties of the communication links used by service directly impact the safety guarantees of the overall system. Common practice often foregoes the rigorous safety arguments about the channel implementation and its interaction with the system components. Hence, it is not clear whether the resulting implementation is correct with respect to its high-level specification.

The key contribution of this work is the first specification of an abstract asynchronous communication channel with explicit support of dynamic creation and tear down of communication links between the network nodes, and its implementation using Java sockets and TCP. For simplicity, our solution associates a unique socket with each communication link between a pair of nodes, and thus it assumes that once a node closes a connection with some destination, it will not try to subsequently reopen it. Our solution can be naturally extended to incorporate multiple, concurrent, point-to-point socket connections. We prove that the implementation preserves the safety guarantees of its abstract specification.

In this work we use the Input/Output Automata model to specify and reason about the behavior of distributed algorithms. A plethora of algorithms have been described using this model. We refer to the language used to describe systems in this model as IOA. It is of practical interest to be able to correctly specify and translate IOA models into executable code.

Tauber [30] wrote the IOA compiler, which uses a target programming framework consisting of Java and MPI The compiler design is proved correct to ensure that the safety guarantees of the source specification are preserved by the resulting Java/MPI implementation. However, the choice of MPI limits the domain of systems to those that do not encounter failures and arbitrary message delivery delays, and that do not have nodes joining and leaving during execution. Given that our approach allows failures, delays, and dynamic node participation, another direct application of the work presented here is an alternative method of implementing robust communication channels using TCP and Java sockets. Note that both methods of communication, i.e., Java/MPI and Java sockets/TCP, may be employed by a compiler, where the first can be chosen for failure-free, performance-oriented applications, whereas the second is chosen for dynamic applications using asynchronous channels.

8.2 Technical development: channel implementation

We present an asynchronous communication channel that connects applications running on any number of networked machines. Each sender node may create connections with any number of receiver nodes, and either the sender node or the receiver node may gracefully close the connection. Messages may be lost, delayed, and delivered out of order. The current model supports only a single socket connection between any two nodes. Thus, once a connection between two nodes is established and subsequently closed, it cannot be reopened (unless it can be determined that the socket can be reused). Allowing multiple, possibly concurrent, socket connections between two nodes is a straightforward extension to this model.

We first defined an automaton, called AbsCh, modeling the behavior of a many-to-many, asynchronous communication channel that allows nodes to spontaneously connect and disconnect. The connections are closed in a graceful way, ensuring that messages that are in-transit are delivered before the connection is closed. The signature, state, and transitions of AbsCh are depicted in Figure 12.

Signature:

Input:
- send(m, i, j), where $m \in M$, $i, j \in I$
- receiverListening(j), where $j \in I$
- senderOpen(i, j), where $i, j \in I$
- receiverStopListening(j), where $j \in I$
- receiverClose(i, j), where $i, j \in I$
- senderClose(i, j), where $i, j \in I$

Output:
- receive(m, i, j), where $m \in M$ and $i, j \in I$
- respReceiverListening(i, j), where $i, j \in I$

Internal:
- senderClosing(i, j), where $i, j \in I$
- lose(m), where $m \in M$

State:
- *messages*, subset of $M \times I \times I$, initially $\emptyset$
- *listening*, subset of I, initially $\emptyset$
- *status* : $I \times I \rightarrow$ {closed, connecting, connected}, initially all closed
- *emptying* : $I \times I \rightarrow$ *Boolean*, initially all false

Transitions:

input send(m, i, j)
Effect:
  if $status(i, j) \neq$ closed $\wedge \neg emptying(i, j)$ then
    $messages \leftarrow messages \cup \{(m, i, j)\}$

input receiverListening(j)
Effect:
  $listening \leftarrow listening \cup \{j\}$

input senderOpen(i, j)
Effect:
  $status(i, j) \leftarrow$ connecting

input receiverStopListening(j)
Effect:
  $listening \leftarrow listening \setminus \{j\}$

input receiverClose(i, j)
Effect:
  $messages \leftarrow messages \setminus \{(m, s, r) \in messages | s = i \wedge r = j\}$
  $status(i, j) \leftarrow$ closed

input senderClose(i, j)
Effect:
  $emptying(i, j) \leftarrow$ true

output receive(m, i, j)
Precondition:
  $(m, i, j) \in messages$
  $status(i, j) =$ connected
Effect:
  $messages \leftarrow messages \setminus \{(m, i, j)\}$

output respReceiverListening(i, j)
Precondition:
  $status(i, j) =$ connecting
  $j \in listening$
Effect:
  $status(i, j) \leftarrow$ connected

internal senderClosing(i, j)
Precondition:
  $emptying(i, j)$
  $\forall (m, s, r) \in messages, s \neq i \wedge r \neq j$
Effect:
  $status(i, j) \leftarrow$ closed
  $emptying(i, j) \leftarrow$ false

internal lose(m)
Precondition:
  $(m, i, j) \in messages$
Effect:
  $messages \leftarrow messages \setminus \{(m, i, j)\}$

Figure 12: Signature, state, and transitions of the abstract many-to-many automaton, ABSCH.

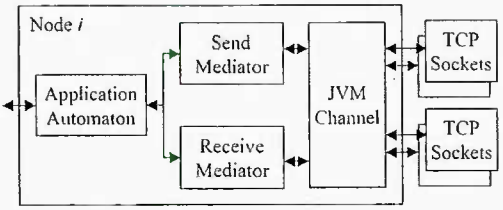

Figure 13: Node automaton.

Next, we developed an automaton, called Jvm-TcpCh, that models the behavior of the Java interface to a communication channel using TCP. We do not model TCP itself or the Java Virtual Machine (JVM) environment; instead, we concentrate on the high-level behavior and the specific interface with sockets via the Java libraries.

Following Tauber's approach [30], we then establish a mediation between the sending application, the communication channel, and the destination application. The mediating automata are mapped to the nodes of the corresponding application automata, as illustrated in Figure 13, showing a node automaton composed of an application automaton and mediator automata, where the mediator automata interact with the TCP sockets through the JVM-TCP channel interface. We refer to the composition of the Jvm-TcpCh automaton with the mediating automata as the CompCh automaton.

The method of forward simulation [16] is used to prove our main result that CompCh implements AbsCh, hence preserving the properties of our abstract asynchronous channel. The full technical development can be found in the available technical report. The main result is formally stated as follows.

Theorem 1 *The set of traces of* CompCh *is a subset of the set of traces of* AbsCh.

9 Conclusion

This results documented in this report were developed under Phase I STTR contract for topic AF07-T019. This project advanced the state of the art in formal modeling and engineering of complex distributed systems. The project included: (a) modeling language that can be used to represent complex distributed systems, theory and methodology providing mathematical basis for modeling systems and reasoning about their properties, (b) extensible and scalable analysis tools that can be used to validate correctness and performance properties, and synthesis tools for producing efficient deployment schemes of the software components in target networks subject to specified constraints. The project extended the methodology to incorporate additional means for reasoning about probabilistic and hybrid systems. The project extended an integrated development environment, called Tempo, for modeling, synthesis, and analysis of distributed systems, developed tools for efficient deployment of the software components in target networks, and explored a methodology

for generating code.

Current work on future extensions for the Tempo toolset and the overall methodology is funded by NSF, and includes work on distributed code generation from Tempo specifications and optimization of distributed system deployment in target network platforms.

Current releases of Tempo toolset for Linux, Windows, and OSX/PPC platforms are available at www.veromodo.com.

References

[1] M. C. Bastarrica. *Architectural specification and optimal deployment of distributed systems.* PhD thesis, University of Connecticut, 2000.

[2] M. Cecilia Bastarrica, Steven A. Demurjian, Alexander A. Shvartsman. Software Architectural Specification for Optimal Object Distribution. *SCCC 1998*, pages 25-31, 1998.

[3] M. Cecilia Bastarrica, Rodrigo E. Caballero, Steven A. Demurjian, Alexander A. Shvartsman. Two Optimization Techniques for Component-Based Systems Deployment. *SEKE 2001*, pages 153-162, 2001.

[4] R. Canetti, L. Cheung, D. Kaynar, M. Liskov, N. Lynch, O. Pereira, and R. Segala. Analyzing Security Protocol Using Time-Bounded Task-PIOAs. *Journal of Discrete Event Dynamic Systems (DEDS)*, volume 18, number 1, March 2008.

[5] Ran Canetti, Ling Cheung, Dilsun Kaynar, Nancy Lynch, and Olivier Pereira. Modeling Bounded Computation in Long-Lived Systems. Submitted for publication, 2008.

[6] Anna E. Chefter. A simulator for the IOA language. Master's thesis, MIT Department of Electrical Engineering and Computer Science, 1998

[7] Oleg Cheiner and Alex Shvartsman. Implementing an eventually-serializable data service as a distributed system building block. In M. Mavronicolas, M. Merritt, and N. Shavit, editors, *Networks in Distributed Computing*, volume 45 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 43–72. American Mathematical Society, 1999.

[8] A. Fekete, D. Gupta, V. Luchangco, N. Lynch, and A. Shvartsman. Eventually-serializable data services. In *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 300–309, 1996.

[9] S. Garland, N. Lynch, Joshua Tauber, and M. Vaziri. *IOA User Guide and Reference Manual.* MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, 2003.

[10] Chryssis Georgiou, Nancy A. Lynch, Panayiotis Mavrommatis, Joshua A. Tauber: Automated Implementation of Complex Distributed Algorithms Specified in the IOA Language. ISCA PDCS 2005: 128-134

[11] Chryssis Georgiou, Peter M. Musial, Alexander A. Shvartsman: Long-lived Rambo: Trading knowledge for communication. Theor. Comput. Sci. 383(1): 59-85 (2007)

[12] Chryssis Georgiou, Peter M. Musial, Alexander A. Shvartsman, Elaine L. Sonderegger: An Abstract Channel Specification and an Algorithm Implementing It Using Java Sockets. *Proceedings of The Seventh IEEE International Symposium on Networking Computing and Applications, NCA 2008*, pages 211-219, 2008.

[13] Dilsun K. Kaynar, Nancy A. Lynch, Roberto Segala, and Frits W. Vaandrager. The Theory of Timed I/O Automata. Synthesis Lectures on Computer Science, Morgan and Claypool Publishers, 123 pages, 2006. ISBN 159829010X.

[14] Kim G. Larsen, Paul Pettersson and Wang Yi. UPPAAL in a Nutshell. In Springer International Journal of Software Tools for Technology Transfer 1(1+2), pp. 134-152, 1997.

[15] Daniel Liberzon, Sayan Mitra, and Nancy Lynch. Verifying Average Dwell Time of Hybrid Systems. To appear in *ACM Transactions in Embedded Computing Systems.*

[16] Nancy Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1996.

[17] N. Lynch, L. Michel, and A. Shvartsman, "Tempo: A Toolkit for the Timed Input/Output Automata Formalism", *First International Conference on Simulation Tools and Techniques for Communications, Networks and Systems (SIMUTools 2008).* Industrial Track: Simulation Works. CDROM, paper 3105, 8 pages, Marseilles, France, March 4-7, 2008.

[18] Nancy A. Lynch and Alexander A. Shvartsman. *A Framework for Modeling and Analyzing Complex Distributed Systems.* Final Technical Report. STTR Phase II Contract No. FA9550-05-C-0178 VeroModo Inc., April 30, 2008.

[19] Nancy Lynch, Roberto Segala, and Frits Vaandrager. Observing Branching Structure through Probabilistic Contexts. *SIAM Journal on Computing*, 37(4):977-1013, September 2007.

[20] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219-246, September 1989. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands. Technical Memo MIT/LCS/TM-373, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, November 1988.

[21] L. Michel, A. Shvartsman, E. Sonderegger and P. Van Hentenryck, "Optimal Deployment of Eventually-Serializable Data Services.", *Proceedings of the Fifth International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR 2008*, Paris, France, May 20-23, 2008.

[22] L. Michel, A. Shvartsman, E. Sonderegger and P. Van Hentenryck "Optimal Deployment of Eventually-Serializable Data Services", Submitted to *Annals of Operations Research*, October, 2008.

[23] Sayan Mitra. *A Verification Framework for Ordinary and Probabilistic Hybrid Systems.* Ph.D Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, September 2007.

[24] Sayan Mitra and Nancy Lynch. Proving approximate implementation relations for Probabilistic I/O Automata. *Electronic Notes in Theoretical Computer Science*, 174(8):71-93, 2007.

[25] Sayan Mitra and Nancy Lynch. Trace-based semantics of Probabilistic timed I/O automata. *Hybrid Systems: Computation and Control (HSCC 2007)*, Pisa, Italy, April 3-5, 2007, volume 4416 of *Lecture Notes in Computer Science*, Springer, 2007.

[26] Peter M. Musial, Alexander A. Shvartsman: Implementing a Reconfigurable Atomic Memory Service for Dynamic Networks. 9th IEEE Workshop on Fault-Tolerant Parallel, Distributed and Network-Centric Systems, pp. 802B (full paper on IPDPS2004 CD-ROM), Santa Fe, NM, 2004.

[27] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In *CAV '96*, LNCS 1102, pages 411–414. Springer Verlag, 1996.

[28] J. Antonio Ramirez-Robredo. Paired simulation of I/O automata. Master's thesis, MIT Department of Electrical Engineering and Computer Science, 2000.

[29] Edward Solovey. Simulation of composite I/O automata. Master's thesis, MIT Department of Electrical Engineering and Computer Science, 2003.

[30] Joshua A. Tauber. *Verifiable Code Generation from I/O Automata for Distributed Computing.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, 2004.

[31] Joshua A. Tauber and Nancy A. Lynch and Michael J. Tsai, Compiling IOA without Global Synchronization, Proceedings of the 3rd IEEE International Symposium on Network Computing and Applications (IEEE NCA04), Cambridge, MA, 2004.

[32] Shinya Umeno. Event order abstraction for parametric real-time system verification. In *International Conference on Embedded Software (EMSOFT 2008)*, Atlanta, Georgia, October 2008.

[33] P. Van Hentenryck, P. Flener, J. Pearson, and M. Ågren. Tractable symmetry breaking for csps with interchangeable values. *Internotional Joint Conference on Artificial Intelligence (IJCAI'03)*, 2003.

[34] *An Extensible and Scalable Framework for Formal Modeling and Analysis, and Development of Distributed Systems*, Proposal to AFOSR, STTR Phase I, Topic No. AF07-T019, Proposal No. F074-019-0196, VEROMODO, Inc., March 2007.

REPDRT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
30-11-2008	FINAL TECHNICAL REPORT	15-SEP-2007 – 14-JUN-2008

4. TITLE AND SUBTITLE

AN EXTENSIBLE AND SCALABLE FRAMEWORK FOR FORMAL MODELING, ANALYSIS, AND DEVELOPMENT OF DISTRIBUTED SYSTEMS

5a. CONTRACT NUMBER	FA9550-07-C-0114
5b. GRANT NUMBER	N/A
5c. PROGRAM ELEMENT NUMBER	N/A
5d. PROJECT NUMBER	N/A
5e. TASK NUMBER	N/A
5f. WORK UNIT NUMBER	0001AC

6. AUTHOR(S)

MICHEL, LAURENT D.
LYNCH, NANCY A.
SHVARTSMAN, ALEXANDER A.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

VEROMODD, INC.
11 OSBORNE ROAD
BROOKLINE, MA 02446

8. PERFORMING ORGANIZATION REPORT NUMBER

VM-07-PHASE1-FINAL-REPORT

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

DR. ROBERT HERKLOTZ
AFOSR
875 N. RANDOLPH STRET
ARLINGTON, VA 22203-1768

10. SPONSOR/MONITOR'S ACRONYM(S)

AFDSR

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

AFRL-OSR-VA-TR-2012-0152

12. DISTRIBUTION/AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTIDN UNLIMITED

13. SUPPLEMENTARY NOTES

N/A

14. ABSTRACT

Report developed under Phase I STTR contract for topic AF07-T019. This project advanced the state of the art in formal modeling and engineering of complex distributed systems. The project included: (a) modeling language that can be used to represent complex distributed systems, theory and methodology providing mathematical basis for modeling systems and reasoning about their properties, (b) extensible and scalable analysis tools that can be used to validate correctness and performance properties, and synthesis tools for producing efficient deployment schemes of the software components in target networks subject to specified constraints. The project extended the methodology to incorporate additional means for reasoning about probabilistic and hybrid systems. The project extended an integrated development environment, called Tempo, for modeling, synthesis, and analysis of distributed systems, developed tools for efficient deployment of the software components in target networks, and explored a methodology for generating code. Releases of Tempo for Linux, Windows, and OSX are available at www.veromodo.com.

15. SUBJECT TERMS

STTR report, modeling language, distributed systems, analysis, simulation, specification, verification, deployment, optimization

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			ALEXANDER A. SHVARTSMAN
U	U	U	SAR		19b. TELEPHONE NUMBER *(Include area code)* 860-486-2672

INSTRUCTIONS FOR COMPLETING SF 298

1. REPORT DATE. Full publication date, including day, month, if available. Must cite at least the year and be Year 2000 compliant, e.g. 30-06-1998; xx-06-1998; xx-xx-1998.

2. REPORT TYPE. State the type of report, such as final, technical, interim, memorandum, master's thesis, progress, quarterly, research, special, group study, etc.

3. DATES COVERED. Indicate the time during which the work was performed and the report was written, e.g., Jun 1997 - Jun 1998; 1-10 Jun 1996; May - Nov 1998; Nov 1998.

4. TITLE. Enter title and subtitle with volume number and part number, if applicable. On classified documents, enter the title classification in parentheses.

5a. CONTRACT NUMBER. Enter all contract numbers as they appear in the report, e.g. F33615-86-C-5169.

5b. GRANT NUMBER. Enter all grant numbers as they appear in the report, e.g. AFOSR-82-1234.

5c. PROGRAM ELEMENT NUMBER. Enter all program element numbers as they appear in the report, e.g. 61101A.

5d. PROJECT NUMBER. Enter all project numbers as they appear in the report, e.g. 1F665702D1257; ILIR.

5e. TASK NUMBER. Enter all task numbers as they appear in the report, e.g. 05; RF0330201; T4112.

5f. WORK UNIT NUMBER. Enter all work unit numbers as they appear in the report, e.g. 001; AFAPL30480105.

6. AUTHOR(S). Enter name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. The form of entry is the last name, first name, middle initial, and additional qualifiers separated by commas, e.g. Smith, Richard, J, Jr.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES). Self-explanatory.

8. PERFORMING ORGANIZATION REPORT NUMBER. Enter all unique alphanumeric report numbers assigned by the performing organ zation, e.g. BRL-1234; AFWL-TR-85-4017-Vol-21-PT-2.

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES). Enter the name and address of the organization(s) financially responsible for and monitoring the work.

10. SPONSOR/MONITOR'S ACRONYM(S). Enter, if available, e.g. BRL, ARDEC, NADC.

11. SPONSOR/MONITOR'S REPORT NUMBER(S). Enter report number as assigned by the sponsoring/ monitoring agency, if available, e.g. BRL-TR-829; -215.

12. DISTRIBUTION/AVAILABILITY STATEMENT. Use agency-mandated availability statements to indicate the public availability or distribution limitations of the report. If additional limitations/ restrictions or special markings are indicated, follow agency authorization procedures, e.g. RD/FRD, PROPIN, ITAR, etc. Include copyright information.

13. SUPPLEMENTARY NOTES. Enter information not included elsewhere such as: prepared in cooperation with; translation of; report supersedes; old edition number, etc.

14. ABSTRACT. A brief (approximately 200 words) factual summary of the most significant information.

15. SUBJECT TERMS. Key words or phrases identifying major concepts in the report.

16. SECURITY CLASSIFICATION. Enter security classification in accordance with security classification regulations, e.g. U, C, S, etc. If this form contains classified information, stamp classification level on the top and bottom of this page.

17. LIMITATION OF ABSTRACT. This block must be completed to assign a distribution limitation to the abstract. Enter UU (Unclassified Unlimited) or SAR (Same as Report). An entry in this block is necessary if the abstract is to be limited.